Distributed Testing of a Device-Level Interface Specification for a Metrology System

John Horst, Thomas Kramer, Keith Stouffer, Joseph Falco,
Hui-Min Huang, Frederick Proctor, and Albert Wavering
National Institute of Standards and Technology (NIST),
Gaithersburg, Maryland, USA.
E-mail: {john.horst, thomas.kramer, keith.stouffer, joseph.falco,
hui-min.huang, frederick.proctor, albert.wavering}@nist.gov

Abstract— **A test suite for a key interface within a dimensional measuring system (coordinate measuring machine or CMM) is presented. The test suite consists of test procedures, test definitions, and various testing utilities. A real-time, distributed test utilizing the test suite has been performed and is described.**

Keywords— **conformance test, coordinate measuring machine, distributed testing, interface specifications, metrology, object-oriented, real-time systems, test suite, validation test**

I. Introduction

Software testing has become critical to software quality. However, interface specification development efforts often treat testing as non-essential. We argue for the early and rigorous application of testing to specification development and implementation, prior to the standardization process. The goal of interface specification development is to create a widely accepted common (or neutral) language at the interface, and the goal of the common language is system interoperability. Following this introduction, therefore, we will briefly describe the interoperability problem and why a common language is desirable.

We will be looking at the following systems on either side of a two-way interface: application software on one side and a real-time logic and motion control device on the other. The device is a coordinate measuring machine (CMM) consisting of device control software (for logic and motion control), motion control hardware, sensors, and actuators. The application software performs many operations including interface to the CMM operator, interpreting inspection programs in higher level languages, analyzing sensor data, and estimating high level features from low-level sensed data.

II. The Interoperability Problem

Automated systems continue to be created for increasingly complex tasks in a variety of industries. As these new systems are implemented, they must be integrated with existing systems. Furthermore, in a competitive market, many systems perform essentially the same task. These competing systems often do not speak the same language either at the input or at the output. Even given the example of a simple two-way interface (see Figure 1), industry is encountering costly problems. The situation often requires the creation and maintenance of up to $2mn$ interpreters (see Figure 2), for m systems of one type and n of another.

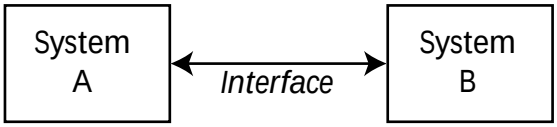

Fig. 1. A simple two-way interface between two separate systems

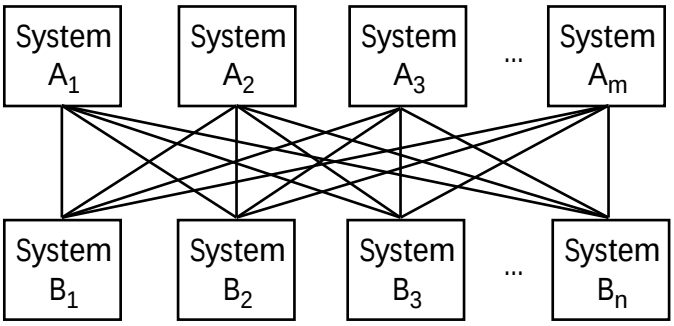

Fig. 2. The same simple interface of Figure 1 with proliferation of up to $2mn$ interpreters required by m different types of system A and n different types of system B. The systems may consist of either hardware or software or both.

System interoperability can be defined as a measure of the ability of multiple systems (all performing roughly the same task) to communicate efficiently and accurately with another set of systems all performing a different task. The failure to achieve interoperability is an ever present and serious barrier to increased productivity and efficiency in industry.

It is perhaps more instructive to look at the interoperability problem from the perspective of a single vendor of an "A" or "B" type system. The m "A" vendors each have up to $2n$ interface language translators to create and maintain, and the n "B" vendors each have up to $2m$ interface language translators to create and

maintain. The total number of interpreters is $2mn$, since only 2 translators total are needed for each of the mn interfaces. One must convert from proprietary language $A_i, (i = 1, 2, \ldots, m)$ to proprietary language $B_j, (j = 1, 2, \ldots, n)$ and *vice-versa*. This is in contrast to the situation where we have a common language agreed to by all vendors. The m "A" vendors each have only 2 interface language translators to create and maintain, and the n "B" vendors also each have 2 interface language translators to create and maintain, or a total of $2(m + n)$ (see Figure 3). Each vendor need only convert from its native language to the common and *vice-versa*. Furthermore, the vendor has little input into the nature of each proprietary language, whereas, if there is a common language, under ideal conditions, every vendor will be able to influence the nature of the common language.

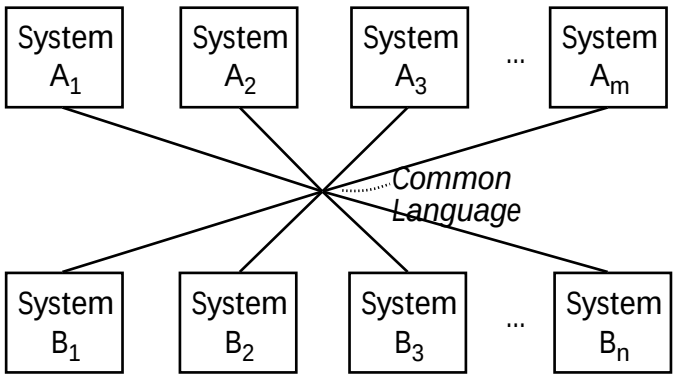

Fig. 3. The simple interface of Figure 1 with proliferation of only $2(m + n)$ interpreters required by all different types of system A and system B due to the agreement on a single common language. If the systems internally employ the same common language, even fewer interpreters are required.

The personal computer (PC) domain reveals this problem very clearly. For example, with different types of central processing units (CPU) and different types of printers, special drivers have to be written and maintained in order to allow different CPUs to interface with various printers and *vice-versa*. This is a substantial drain on resources and furthermore does not successfully solve the interoperability problem. Slowly updated and inaccurate drivers create a quality and efficiency problem.

Potential resolution of the interoperability problem comes in three forms, 1) enforce the use of only one vendor or 2) develop or select a single new language that will be used by all or 3) translate proprietary languages into a single new language, but employing information modelling languages and accompanying tools to facilitate language development. The first type of resolution to the problem, is illustrated whenever an organization mandates the use of a word processor from a single vendor. An example of the second is when certain international conference organizers agree to use English as the medium for communication, an example of "selecting" an existing language. More challenging is to develop a new language for an interface. Examples of information modelling languages which can be used to develop a new common language are Extended Markup Language (XML), Interface Definition Language (IDL) [1], and EXPRESS [2]. These will effectively eliminate the need for the definition of syntax in the specification, since syntax is already part of the modelling language. Neither the first nor the third type of resolution to the interoperability problem has been chosen by the industry, so the second attempt at resolution is assumed in this paper.

III. Background

The CMM has become ubiquitous in manufacturing, particularly in the automotive and aerospace industries, in part due to the importance of accurate dimensional measurement to manufacturing quality. The general application domain of the CMM is called "dimensional metrology," which is simply the science and technology of accurately measuring selected geometrical features on a physical object and making comparison with required tolerances.

The National Institute of Standards and Technology (NIST) helps facilitate solutions to system interoperability problems of industry. About two years ago, it was widely acknowledged that a common language solution to the interoperability problem for the CMM device interface was of high priority. Several automobile and aerospace manufacturers (users) strongly encouraged some CMM hardware and software manufacturers (vendors) to create such a language for the device interface. Several proprietary languages had been developed and used on single-vendor equipment, so the semantical knowledge and language development experience exists. However, standards development and specification (language) testing were weak links. NIST was asked to take a leadership role in these latter domains.

Two common (non-proprietary) device specifications[1] have since emerged and are being examined [3][4], the "CMM-driver" and "I++" specifications, respectively. The two specifications cover exactly the same logical interface and, furthermore, are mostly similar in the kinds of commands and responses chosen for the interface [5]. It now appears that the two specification development teams have agreed to work together to create a single specification using the I++ specification as the baseline. However, up until now, NIST has primarily been working with those responsible for the CMM-driver specification, developing and implementing testing procedures and tools. This document describes what those procedures and tools are, what tasks they perform, and what remaining work needs to be done. NIST has now been asked by the industry to apply the same concepts to and develop similar tools for the I++ baseline specification and we are just beginning this process.

[1] A specification is a standard under development prior to formal acceptance by a broad community of interested parties.

IV. CMM Device-Level Specifications

The CMM device-level interface specifications currently under development [3][4] consist of requirements for communications protocol, software execution paradigm, and the syntax and semantics for command and response across the interface. For example, communications protocol might be TCP/IP or direct serial link. Software execution paradigms might include cyclic execution of commands and responses, preemptive execution, or a combination of both. Syntax may require ASCII text format and the type and ordering of allowable characters. Semantics will specify the precise meaning of commands and responses. This includes definition of the allowable error types and what each error type means.

The audience of the specifications is the technical implementors who will be creating interpreters for application software or device software that either convert from the native language to the common or *vice-versa*. It is expected that users will not be involved in many of the details of the specification. However, users may need to interact with the CMM system directly, not just through the application software, and the interface under consideration is between the application software and the CMM. However, functional requirements as well as requirements about implementation are essential inputs from the users.

There are a multitude of decisions to be made on functionality alone. One of the key issues, for example, is whether to allow uncompensated measurement point data on the interface. The advantage of disallowing uncompensated data on the interface is that responsibility for temperature and thermal compensation of the data lies squarely with the CMM vendor. Also, the CMM vendor is most often the one that can best perform such compensations. The advantage of allowing uncompensated data on the interface is that 3^{rd} party application software can now perform compensation, allowing a broader selection of solutions.

A particularly controversial issue has been whether to allow raw (uncalibrated) probe sensor data on the interface. There are more 3^{rd} party application software packages that perform probe qualification (or probe calibration) than there are 3^{rd} party application software packages for temperature and thermal compensation.

V. The Need for Testing

A specification may be written so as to be ambiguous in its meaning; this is particularly true of natural language specifications, of which [3] and [4] are instances. Since an implementor may choose the wrong interpretation of something in the specification, it is particularly important that the specification be as unambiguous as possible. Even a demonstrably unambiguous specification is still not enough to make for a high quality specification; there needs to be sufficient functionality allowed in the specification. This leads to the need for validation testing which will be dealt with later. There also needs to be some minimization of complexity in the specification, and implementation must be kept as simple as possible, since broad adoption of the specification is essential to interoperability. A high quality specification is a necessary but not sufficient condition for achieving the ultimate goal of high quality interoperability; ensuring high quality implementations is the next step toward that goal.

Even if the natural language specification is unambiguous, it can be misinterpreted by an implementor. Therefore, there must be tests that verify whether the implementation actually conforms to the specification. This is called conformance testing and, if done well, is an important step towards ensuring a high quality implementation.

Several other items are needed to ensure high quality interoperability. If two implementations conform to two different conformance levels[2], a naive user may think two implementations are completely interchangeable because they comply to the same specification. However, they are not interchangeable, because they comply to different conformance levels. In this case, interoperability is not achieved. Therefore, the number of conformance levels should be kept as small as possible. Furthermore, even if a specification is a good one, if there is incomplete agreement and partial acceptance of the specification, or of the testing procedures and tools, high quality interoperability will not be achieved. Finally, it may have taken so long for the specification to be successfully developed and accepted that, even if interoperability is achieved, current technologies may have advanced beyond the technology assumed by the specification. Clearly, high quality interoperability is a challenging goal, requiring all the resources of many organizations; much education on the importance of high quality and timeliness is vital.

It is important to discover the logical separability of software elements in an implementation and testing tools. If we can minimize the elements related to the proprietary implementation of the specification and use common software for everything else, we reduce variability in the testing, which reduces development time and ultimately increases the chance of high quality interoperability. For example, the command objects and its queue have to be used by all implementors. If all testing participants use the same command objects and command queue for their tests, we reduce the size of the domain where implementation errors occur. This makes testing more efficient.

Also, the use of testing allows the application of quantifiable metrics. Quantifiable metrics are needed to measure the success of each implementation and ultimately to ensure interoperability. Testing later will cause more problems (and delays) when the cost to change (after publication of the specification) is substantially higher.

[2] A conformance level is a subset of the total specification

Ultimately, early and thoughtful testing facilitates a high quality, timely specification.

The success of an interface specification is dependent on whether it leads to compatible implementations and whether the specification has sufficient functionality. It would be ideal to express the specification in a formal language, like first order logic, as a set of requirements, and be able to test all implementations automatically against these requirements. Formal methods for describing syntax are available, *e.g.,* Backus Naur Form (BNF)[6], but formal methods for describing semantics (*i.e.,* the meaning of something), are not mature enough to allow one to define all the semantics needed for inspection. Furthermore, specifying requirements and specifications in a formal manner requires that all implementors be familiar with the formal method, a daunting learning curve. This being the case, unambiguous natural language specifications coupled with well-defined testing procedures and testing tools is essential for the success of a specification.

This is like the situation with formal computer languages. The language itself is defined in natural language, but the compiler acts like a testing tool to check for inconsistencies and errors in the syntax. However, semantic errors are not caught by compilers. We would want a testing procedure to catch all syntax errors and as many semantic errors as possible.

Syntactical errors can be detected "informally," as long as the testing tools exhaustively look for and flag such errors. This is not too hard to do. Much more challenging is how a testing procedure detects semantic errors. It appears to be impossible to test for all semantic errors, but some obvious problems can be detected. For example, if a CMM system responds to a measure point command without either returning measured points or an error, we have a semantic problem, even if all syntax is correct.

VI. NIST Involvement in Testing

NIST's role in the specification development process is primarily to develop and maintain testing tools for use by the industry. NIST's involvement consists of the following,

1. develop and maintain the entire testing suite
2. maintain an issues log to keep track of necessary changes to the specification and the test suite
3. provide a leadership and advisory role in the planning and conduct of various meetings and specification review efforts
4. perform those tests that will be necessary to improve the testing suite

A local testbed will be maintained (see Figure 4). Test analysis tools may be developed by NIST and some limited analysis may be done at NIST. This type of involvement by NIST has a long history in the development of many standards such as Dimensional Measuring Interface Standard (DMIS) [7], XML, and Standard for the Exchange of Product Model Data (STEP) [8].

VII. Test Suite

We define the total collection of all testing related entities as the "test suite." The entire test suite consists of the following elements, most of which are under varying stages of development at NIST.

- test types
  - validation tests
  - conformance tests
    - CMM implementation tests (Figure 5)
    - application implementation tests (Figure 6)
    - cross testing (simultaneous testing of two implementations, as shown in Figure 7)
- testing utilities
  - test cases
    - inspection plans
    - test artifacts
  - common test software
    - application simulator
    - response simulator
  - command and response classes
  - analysis tools and metrics
- testing procedure

The test suite ostensibly checks to make sure an implementation is written according to the specification. However, the test suite is ultimately meant to improve the specification. As we conduct the tests, we will invariably find that certain aspects of the specification need adjustment. As we perform and analyze tests, we will also discover that the test suite itself need to be improved. So, testing is iterative with the implementation development, the specification, and the test suite itself.

Each of these elements in the test suite will now be described.

VIII. Types of Tests

Conformance testing is the effort to determine the level of compliance of implementations of the specification to the specification itself, *i.e.*, does the implementation comply with the specification. Validation testing is the attempt to guarantee that all the functionality needed on the interface is explicit in the specification.

Validation testing should be done early in the specification development process, so that all necessary functionality is being integrated into the specification. In the same manner, conformance testing should begin early in the development process and be used to test every new implementation of the specification. The validation test will not have such continued use. For a good overview of conformance testing see [9].

A. Validation Tests

Validation testing is necessary to assure the developers that all the functionality required by users of the interface is sufficiently expressible in commands and responses detailed in the specification. The two specifications under discussion [3][4] have not, as far as we know, undergone any formal validation tests for functionality.

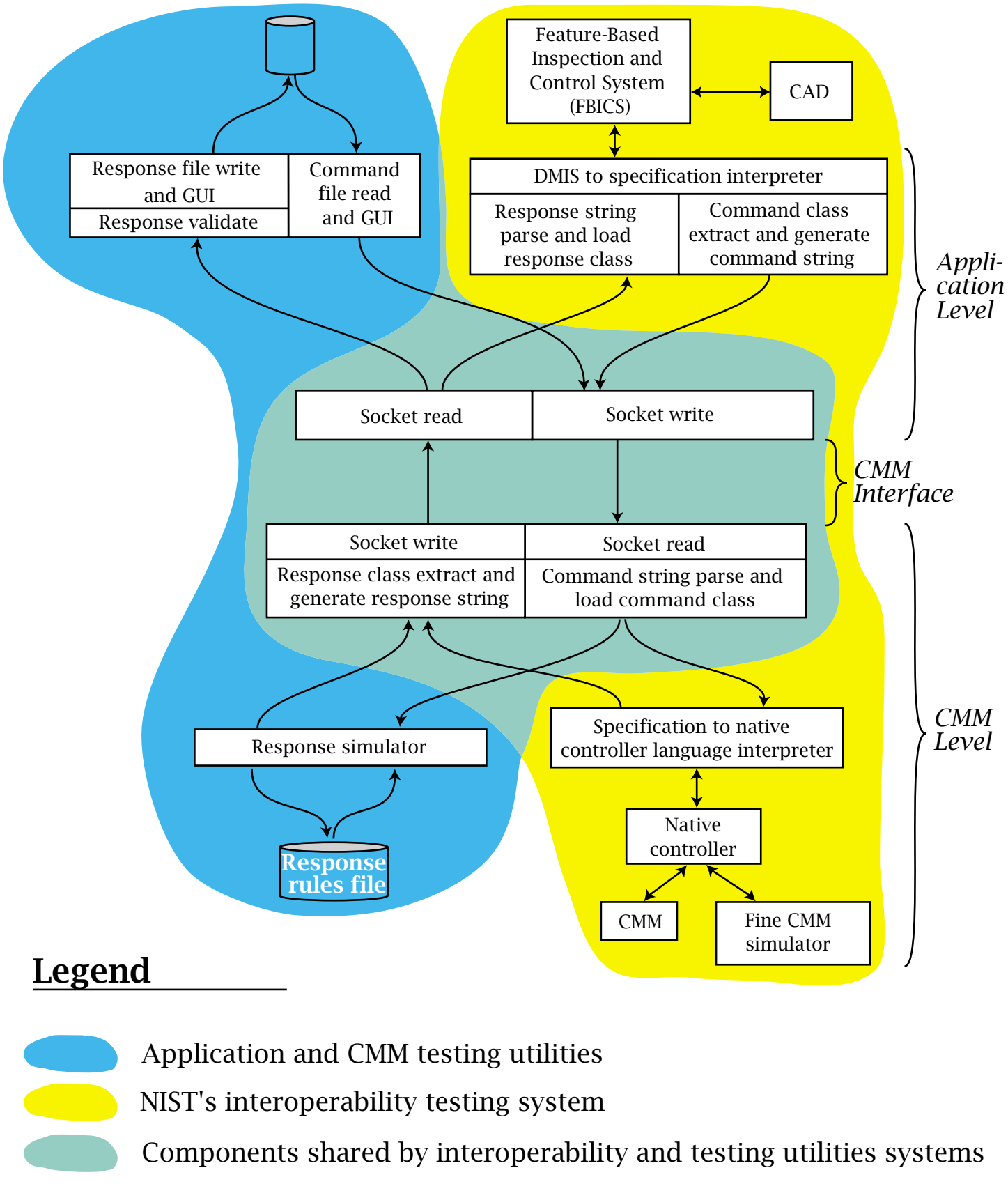

Fig. 4. NIST CMM Interface Specification Testing System: This graphic breaks down into components both the application and CMM utilities as well as NIST's own application system and CMM system. It also reveals the distinct use of object-oriented command and response classes in the overall testing architecture.

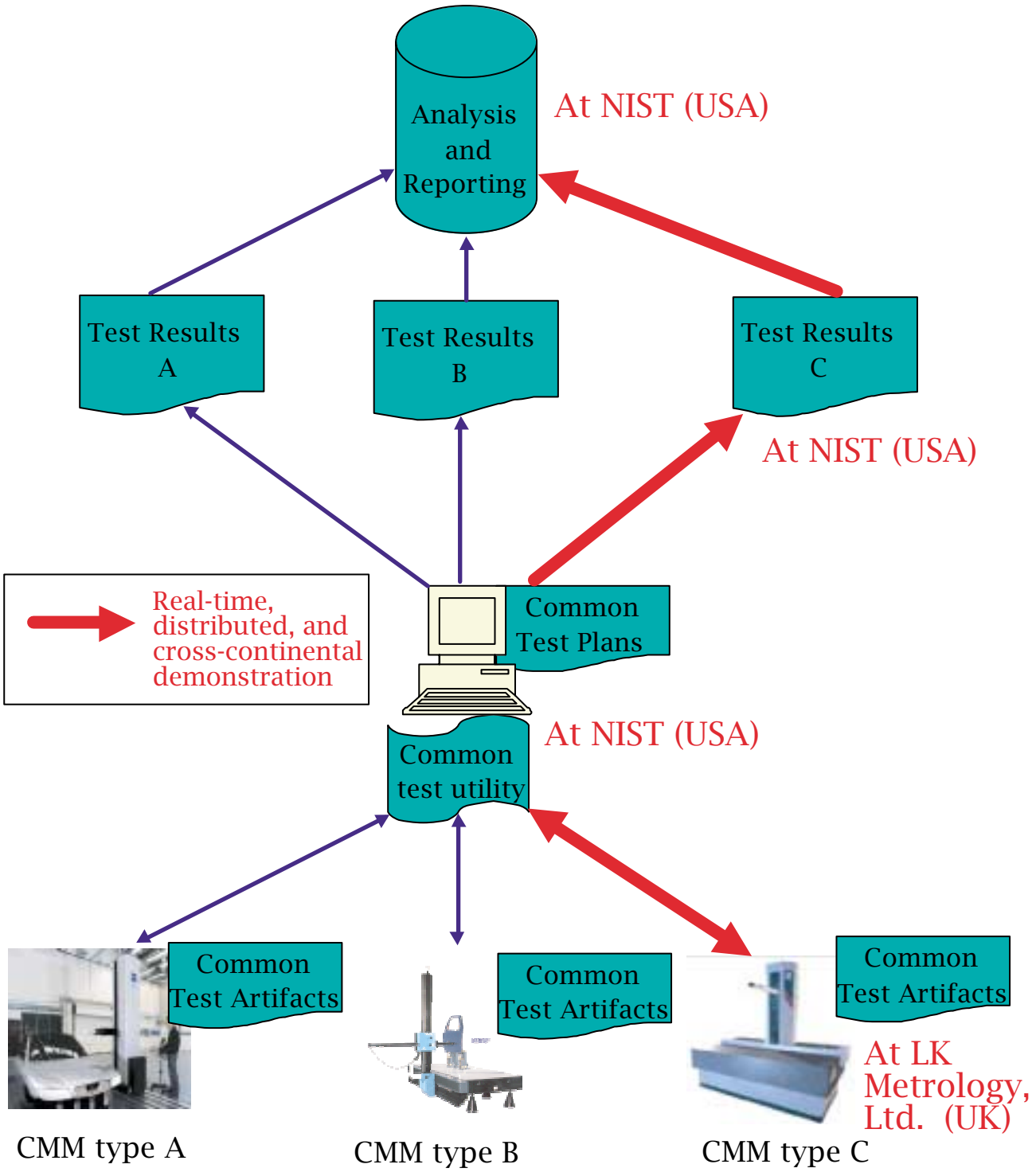

Fig. 5. A graphic describing a set of tests of the conformance to the specification of three CMM implementations of the specification employing various utilities in the test suite.

Application Implementation Conformance Test

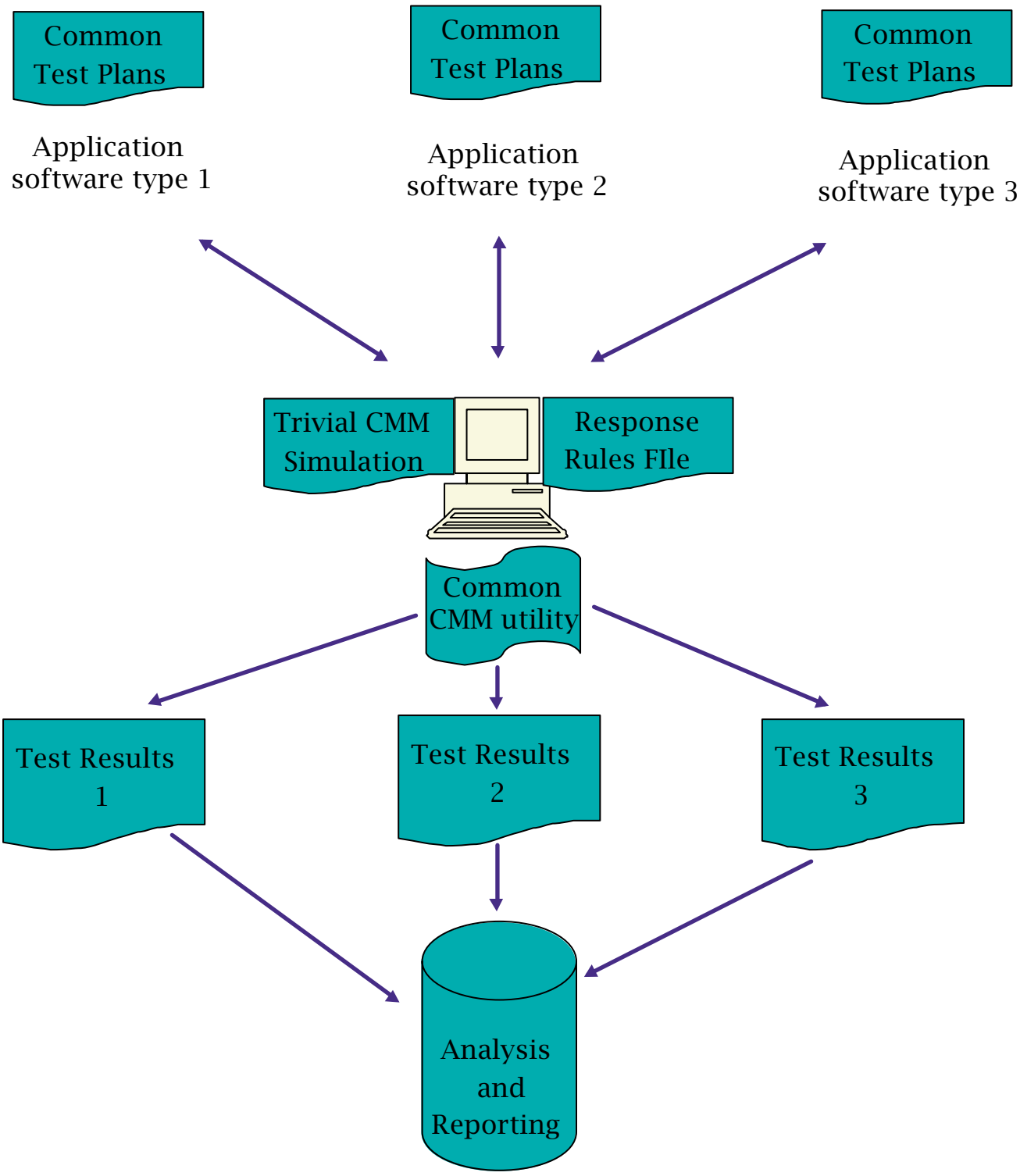

Fig. 6. A graphic describing a set of tests of the conformance to the specification of three application software implementations of the specification employing various utilities in the test suite.

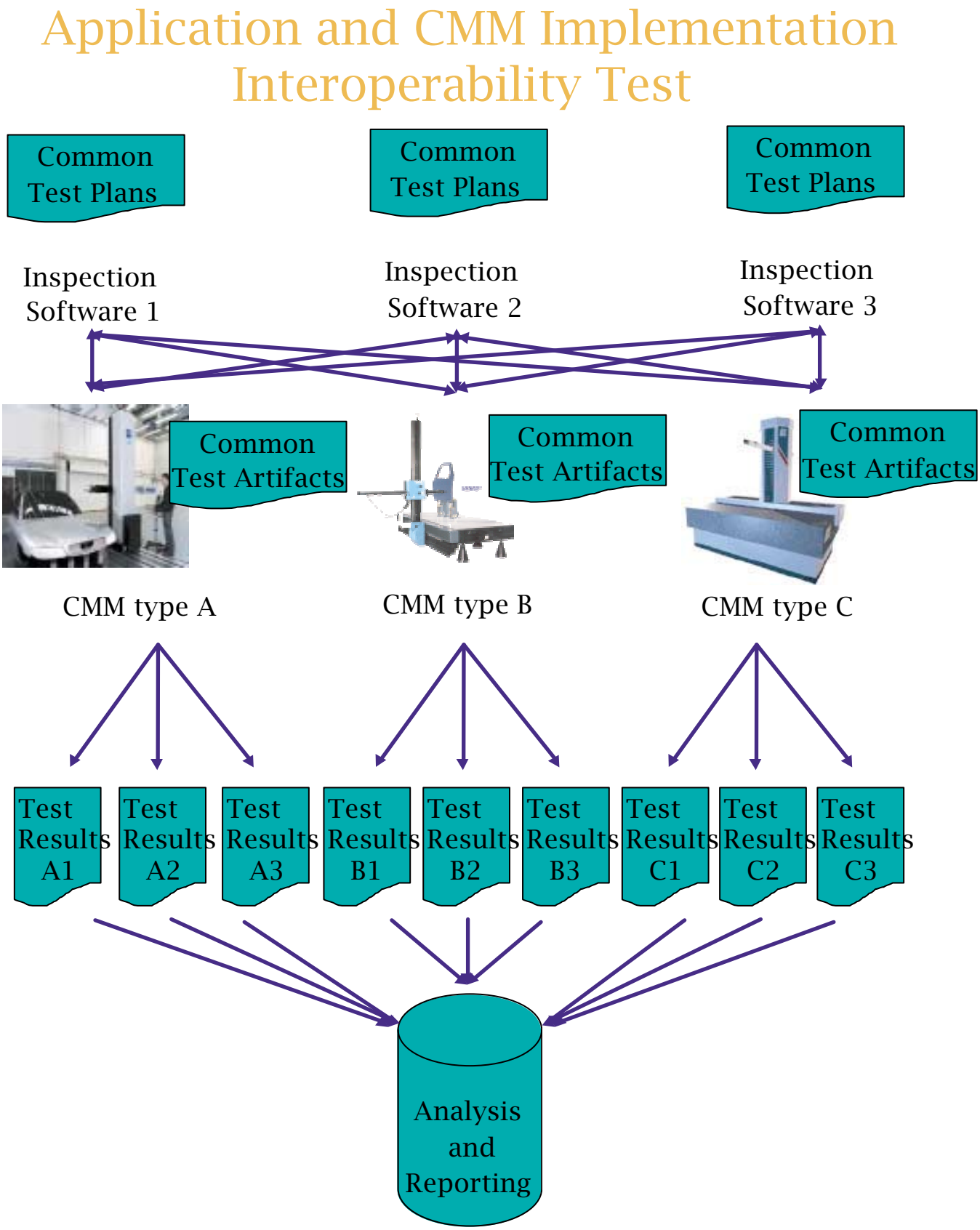

Fig. 7. A graphic describing a set of tests of the level of interoperability of three application software implementations of the specification with three CMM implementations of the same specification.

Functionality "coverage" has been accomplished informally (as thought experiments).

B. Conformance Tests

A useful family of conformance tests for the specification might consist of three classes of tests. The first two are conformance tests and the last is an interoperability test.

1) A suite of test files containing specification compliant strings are placed on a socket by the common application utility (see Section X-A.3) and sent to a CMM simulator or a real CMM that has an implementation of the specification. The test will automatically (or manually) compare the real output to the expected output and log the results of this comparison for analysis. The test files will command the CMM to perform various inspection tasks on a common test artifact. Test files include erroneous commands to test for proper response to such inputs. This is done for n CMM systems. Figure 5 gives an example of this type of test. The thick arrows in the figure correspond to a real-time, distributed, cross-continental test that was performed on July 13, 2001 as described in Section XIII.

2) A suite of test response files are generated for use by the (common) CMM utility software (see Section X-A.4). The response files will include a variety of rules for producing various kinds of responses depending on the input received by the CMM utility. The rules will look at the commands, command sequence, and random events to determine the appropriate response for each test. The response files will contain rules for producing legal and illegal responses for both semantics and syntax. Response files coupled with the CMM utility will be employed with all m application software packages. Figure 6 gives an example of this type of test. High level test inspection plans may be used but may not be necessary or even helpful. The CMM utility will automatically (or manually) compare the real output of the application software (subsequent commands) to the expected output and log the results of this comparison for analysis. The goal in testing application implementations is to minimize (or avoid altogether) any required testing modules in the application software.

3) A suite of test files consisting of high level inspection plans are input to an implementation of the specification by an application software package that performs inspection plan execution and sends it to a CMM simulator or a real CMM that has an implementation of the specification. Figure 7 gives an example of this type of test. The output of the application software will also be sent into the common receiver utility, and comparisons and results will be logged for analysis. The test files (again including errors) will command the CMM to perform various inspection tasks on a common test artifact. This test will be repeated for the mn combinations of m inspection plan execution software packages and n CMM systems. An analysis utility will compare the real output (subsequent commands) of the high level inspection program to the expected output and log the results of this comparison for analysis.

The goal of performing these tests is to verify the correctness of implementations, produce a complete and unambiguous specification, and facilitate interoperability.

IX. NIST Testbed

In order to develop and thoroughly test all elements of the test suite, it is necessary to have a testbed containing whatever is needed to perform all the types of validation and conformance tests (presented in Section VIII) for at least one application software system and one CMM system. Such a testbed exists at NIST and is shown in Figure 4. This figure breaks down into components both the application and CMM utilities as well as NIST's own application system and CMM system.

The interoperability testing system at the application level can allow the user to enter commands to measure features on a part at a high level. This is the Feature-Based Inspection Control System (FBICS) [10], which interacts automatically with Computer-Aided Design (CAD) software to produce DMIS commands at the output. A DMIS interpreter converts DMIS into device-level interface commands (and converts responses to DMIS). Currently these commands and responses are in NIST's own native language, but can easily be converted to whatever device-level specification industry agrees to support.

The conformance testing system involves both testing utilities and the interoperability testing system (application and CMM levels). For testing the application utility, testing would be performed with both the response simulator and the CMM system (both simulated and real). For testing the CMM utility (the response simulator), testing would be performed with both the application utility and NIST's application software system.

X. Testing Utilities

Successful design and implementation of the validation and conformance tests requires that there be a mix of testing utilities to support these tests. The general idea is to have the same (common) utilities used by everyone conducting tests. This minimizes variations in the meaning of the test results. If errors in the test utilities occur, all testing participants should experience them and the common testing utilities can be fixed. Testing utilities include the following:

- test cases
  - test artifacts
  - inspection plans
- common test software
  - application simulator
  - response simulator
- analysis tools and metrics

We need to test implementations on both sides of the interface. Interestingly, such tests are not symmetric.

Tests for implementations on the CMM side simply require a sufficient set of test cases, each consisting of a test artifact paired with a test inspection plan for that artifact. However, testing implementations on the application side are more subtle and challenging. Essentially what is required is to create a response simulator and some sort of analysis tool that examines the response of the application software to the response simulator. The analysis tool must be independent of the application software and not require any compilation of the analysis tool into the application software. The response simulator must be able to generate both correct and incorrect responses of sufficient variation to fully exercise the application's implementation of the specification. Furthermore, we may want to require a set of high level inspection plans to test the application's implementation. However, the application software will invariably translate high level commands into low level commands uniquely from other implementations, making it difficult to do quantitative test and analysis of results.

We have not created application software testing utilities as yet, except that we have a CMM response utility that parses specification compliant commands and very roughly simulates CMM-like responses. Simulation involves simply appropriate delays for "move" and "measure" commands. We now describe tools for testing CMM implementations of either specification.

A. Test Cases

A single test case consists of a test artifact paired with a test inspection plan for that artifact. We want these test cases to provide sufficient coverage of the types of commands allowed in the specification. We use these test cases to sufficiently exercise the commands from the specification as realized in each implementation under test.

A.1 Test Artifacts

Common test artifacts are needed to minimize the sources of error in the various implementations. We need to be able to ensure that any problem in execution of a test inspection plan is not attributable to the test artifact. If we restrict all tests to specific and common test artifacts, we stand a better chance of efficiently debugging problems in the various implementations of a specification and ensuring that the implementation conforms to the specification.

Neither specification contains any single command intended to measure a high level feature (like a cylinder). The commands are at a low level of abstraction, like "measure a point" or "go to a point." Combinations of these low level commands are intended for use to measure high level features, but such aggregation would form part of a higher level specification such as DMIS. Therefore, it would seem that features need not be explicitly defined in any test artifact. However, a variety of types of measurements and moves of the probe and head is required to fully exercise either specification. This argues for a test artifact with a small variety of features. Three non-parallel flat surfaces are necessary for localization of the artifact. A spherical surface is helpful to test simultaneous moves of CMM arm and head and potentially, scanning-type measurements. Cylindrical features at several orientations are also useful to test discrete moves of the CMM arm and head.

The artifact needs to be quickly available and inexpensive, since we expect many implementors to be involved in the testing process. Furthermore, it is expected that certain implementors will get involved at different times in the testing process, and we want to assure them that they can begin testing with the others as soon as possible.

Neither specification requires a highly accurate artifact, since no test of the specifications would require any unusual accuracy or repeatability of measurements. This said, the part must be accurate enough. Substantial inaccuracy in the assembly could have negative consequences. For example, there must be three non-parallel planes on the artifact, which are used to determine the artifact coordinate system (three such planes can be seen in the tall tower in the lower right hand corner of Figure 8). If these planes vary significantly from artifact to artifact, a subsequent measurement at the extremity of the artifact may be so far off that an expensive CMM probe might be damaged.

These requirements led us to choose an artifact assembled from freely available Lego™[3] building block pieces. This approach meets all the required constraints. Example assembly instructions have been generated for the artifact. CAD model of the assembly is shown in Figure 8.

In order to avoid grossly inaccurate artifact dimensions, a test artifact with a stiff assembly is required. The first step is to create a stiff layer forming the foundation for the artifact. The concept of stiffness is also applied to all subassemblies of the artifact in 8). The "algorithm" used for both the foundation and all subassemblies is simply to maximize the occurrence of "crossings" from layer to layer while using the largest sized piece possible. A crossing is when the outside boundary edges of the top piece has minimal coincidence with the outside boundary edges of the pieces below.

A.2 Inspection Plans

Initially, NIST modelled the test artifact in STEP AP224 [11][4]. We used the FBICS [10] to generate DMIS code to inspect a few features on it. The idea was that if

[3]Certain commercial products are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply any judgement by the National Institute of Standards and Technology concerning these products, nor is it intended to imply that they are necessarily the best available for the purpose.

[4]AP stands for Application Profile

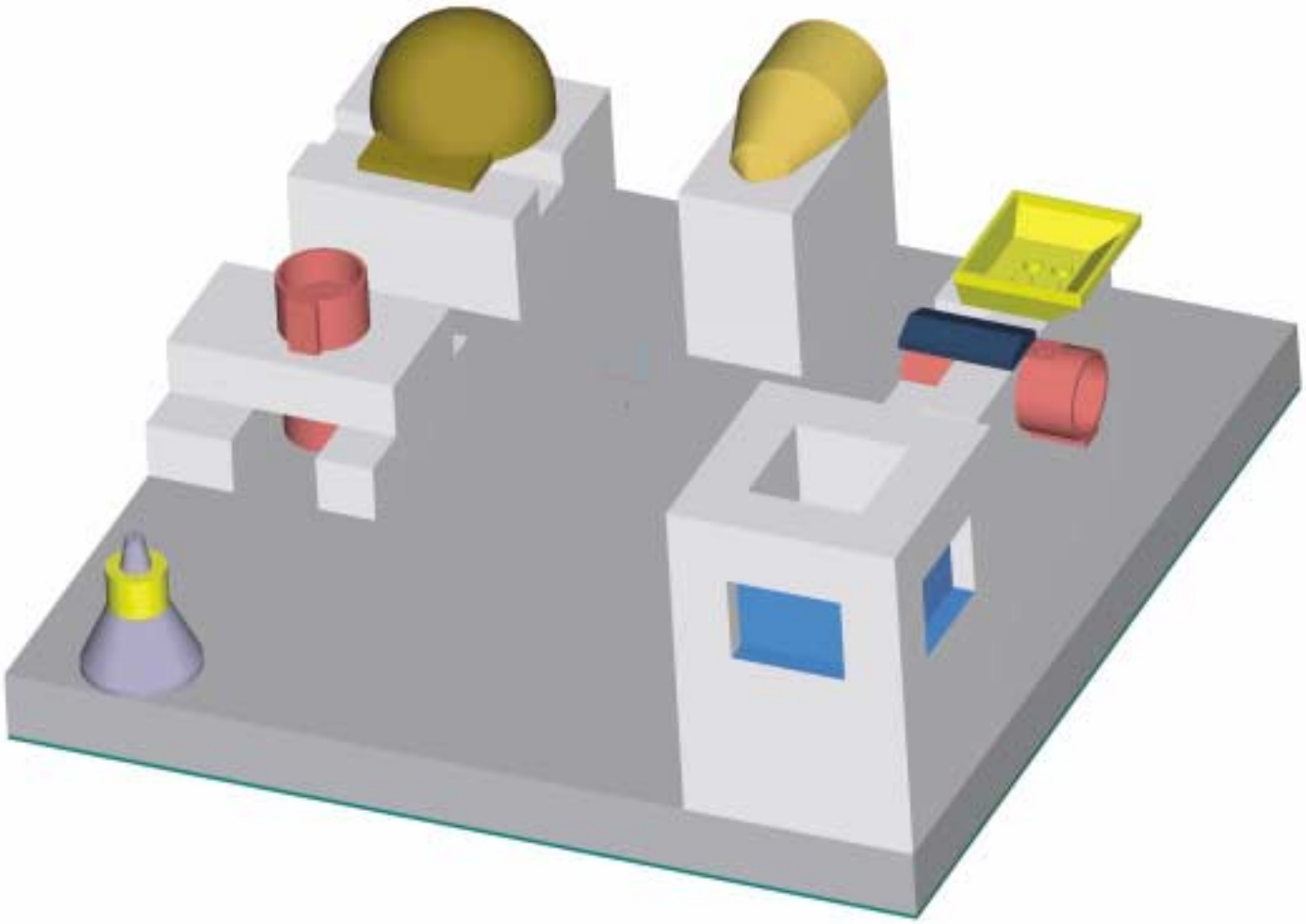

Fig. 8. CAD model of test artifact

NIST's DMIS interpreter were modified to generate CMM-driver compliant commands, the command file could then be generated automatically. The DMIS interpreter was only partially so modified because the output file of device-level inspection commands produced by the existing interpreter was much more complex than necessary. So it was ultimately easier to write the command file by hand.

Eventually, we expect that the test inspection plans will consist of files of both high level (*e.g.*, DMIS) commands and files consisting of lists of specification compliant strings. We must have the latter in order to successfully introduce errors into the list of commands. A subset of these files will simply contain one command per file. Log files consisting of the correct responses will exist for each file. The inspection plans should not require any artifact other than the test artifact(s). We want the files to test both syntax errors and semantical errors. The latter would include errors in execution, for example, the probe being sent to measure a non-existent point or the probe encountering an obstacle prior to an approach point in a "measure" command or in a "move" command.

The file format for files of CMM-driver commands is based heavily on the file format created by D. Smith of LK Metrology, Ltd. [3] in the form of a set of sample files of CMM-driver commands that he used for testing.

Some of the files we wrote, after debugging, were error-free, some were filled with intentional errors, and some had only a smattering of intentional errors.

The error-free files included:

1. Some similar to what a list of CMM-driver specification compliant messages coming from a DMIS interpreter might look like when DMIS code was being interpreted for inspecting a simple DMIS feature such as a cylinder or a plane.
2. Some that simply tested part of the CMM-driver specification by including all messages of a certain type (such as all messages for getting and setting parameter values).

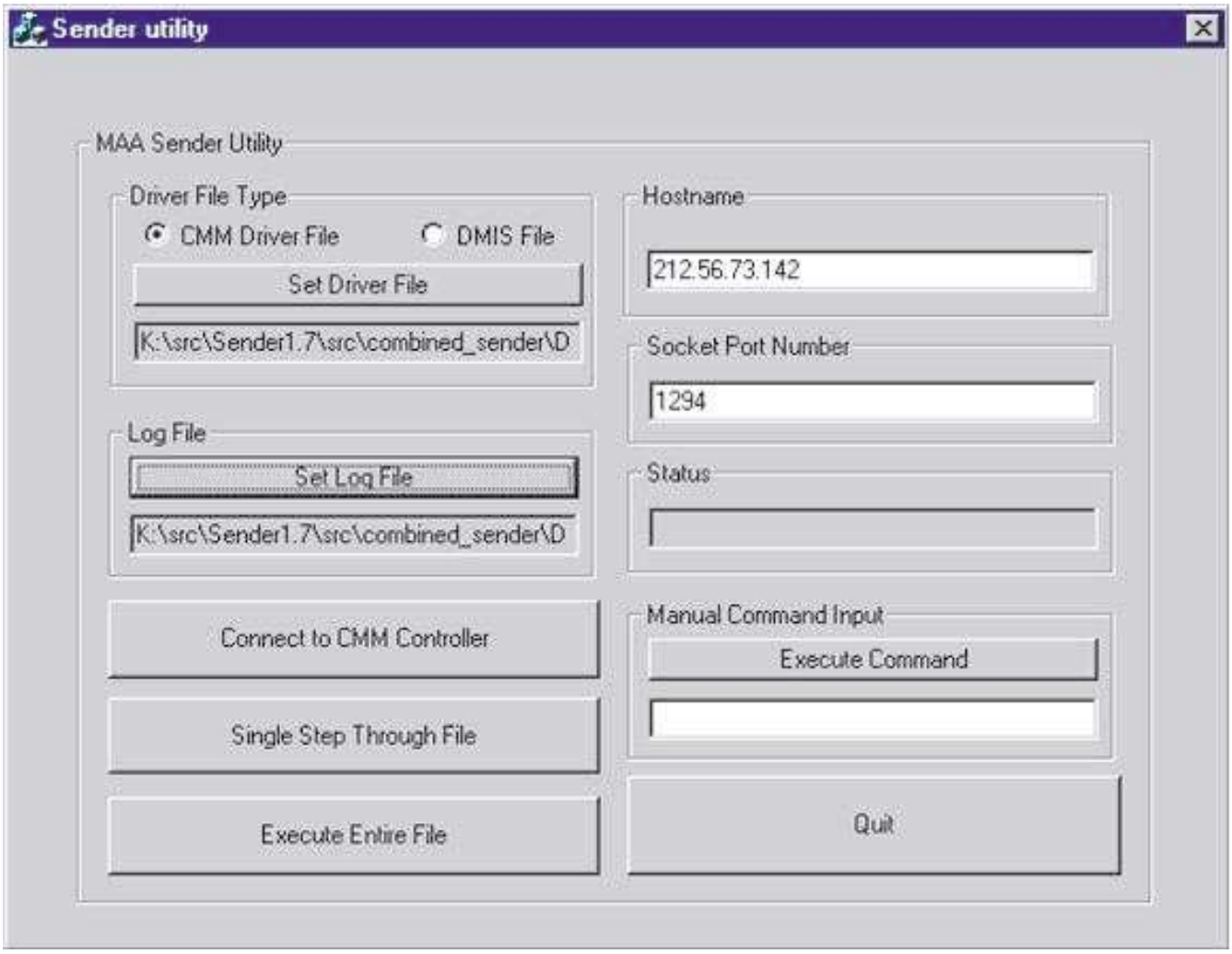

Fig. 9. Application utility GUI

The error-filled files were of two types:

1. Sets of command messages with various syntax errors. Files of this sort may contain an error on every line.
2. Sets of command messages that cause execution errors. This type of file necessarily contains about two-thirds correct commands, since most execution errors can only occur when a particular machine state has been reached, and it usually takes two or three correct commands to reach a desired state.

For each command file, we wrote a corresponding trace file. The trace file is similar to what would be expected to be in a log file prepared by the application utility. The file consists of pairs of lines, the first being a line from the corresponding command file, and the second being the responses that would be expected from a system executing the command. A real log file, however, would have only one response per line and would not be so nicely ordered, since queueing is used (resulting in "done" messages arriving much later). Also, in the hand-written file, all probed points are at the exact nominal location, whereas in a real file, it would be surprising if any point were exactly at the nominal location.

We wrote a test program (CMM-driver specification compliant command file) for the test artifact that was used in an international demo with LK Metrology, Ltd. The demo is described in Section XIII. D. Smith of LK Metrology, Ltd. [3] added substantially to this program. In connection with the test program we wrote C++ software for translating goal points in a CMM-driver specification compliant command file. This was needed since the test artifact might be located on any part of the table of the CMM doing the inspection.

A.3 Application utility software

An application utility has been fully developed for distributed testing of the CMM-driver specification defined in [3]. The application utility is a graphical user interface (GUI)-based, object-oriented program that runs on a personal computer (PC) platform and was developed in Visual C++. The commands are sent over a TCP/IP socket to a specification compliant controller. The controller receives the commands, executes them, and returns the appropriate response back to the application

utility via the socket. The application utility creates a time-stamped log file of commands sent and received over the socket and performs a validation test on the returned responses to determine if the CMM controller is compliant with the specification. The application utility and a small set of test command files (inspection plans) were delivered to several CMM vendors for evaluation and testing of their specification compliant controllers.

The user executes the application utility through the GUI shown in Figure 9. The user first selects which type of file to run and the name of the file to run. The application utility can run either of two types of files, a low-level command file or a DMIS file. When a DMIS file is selected, the file is run though an interpreter that converts the DMIS command to the appropriate specification compliant low-level command(s). Only a subset of DMIS commands are supported at this time. The user then selects the name of the log file where the time-stamped data will be recorded. The user specifies the host name of the controller. The host name can be entered as either a fully qualified host name or IP address. The user then specifies the port number. The default port number is 1294 as specified in [3]. When the user pushes the "Connect to CMM Controller" button, a non-blocking TCP/IP socket is created between the application utility and the CMM controller on the specified port. Once the application is connected to the controller, the user can either enter a command manually, single step through the program file that was selected, or run the entire file. A status window displays the current status of the executing program, including what command was just sent or received and any error conditions that exist.

A.4 CMM utility software

The CMM utility software will eventually allow an application software vendor to test his or her implementation of either specification. The idea is to have a set of files consisting of response rules that given a specific event will output a specific response or sequence of responses. The event that triggers the response may be a specific command, or a specific command sequence, or the tick of a clock, or some random event. The utility will examine the nature of the subsequent commands from the application, looking for expected command sequences.

We may also want to create high-level test inspection plans, *e.g.* DMIS, for use with the CMM utility test software. However, the problem here is as follows. First, we cannot force the generation of certain low-level commands (such as "get parameter") with the detailing of a high-level command. The application developer, in his or her test measurement routines, may never test certain low-level commands. Certain response rules will never fire because the events needed to fire those rules never occur. This will be a complete test of an implementation only if those particular commands are never needed by the application, which is doubtful. Secondly, the particular low-level commands (and the order of execution of those commands) accompanying a particular high-level command will be to some degree the unique choice of the application software developer. On the other hand, certain high level commands, *e.g.*, to "measure a point" or to "go to a point" will of necessity require a low-level "measure" and/or a "move." This issue needs to be debated and resolved as the metrology industry moves along the specification development and testing process.

The current implementation of the CMM utility includes a command parsing engine coupled with both a real CMM and a simple CMM simulator. NIST developed CMM-driver specification compliant message parsing software, which was integrated into the CMM utility software. The source code, along with the executable code, for the CMM utility is available to all testing participants.

The current CMM utility is expected to interact with the application implementation in terms of receiving and handling CMM-driver specification compliant CMM control commands from any application and generating appropriate responses. The commands are to be coded as ASCII character strings and sent through a communication socket with a mutually agreed port number.

This CMM utility software provides the following command reception functions:

- Read text strings from the pre-designated communication socket.
- Interpret the strings according to the specification and either extract for command information or determine error severity and report the errors.
- Manage the commands, *i.e.*, either place them in a queue or abort them.

Both specifications detail the syntax and semantics for error responses. For example, the CMM simulator produces errors such as "parameter out of range" and "illegal probe type." In order to be able to execute these error responses and to be able to verify the correctness of the command reception, the software provides the following CMM simulation functions:

- Retrieve the commands from the queue. Execute them using a state machine model.
- Simulate the machine behavior, in low fidelity, to provide feedback for command execution.
- Report errors if execution fails.

The CMM utility executes at a uniform rate, *i.e.*, cyclically. Its first function is to read the socket for command strings. A parser processes the command string into command serial number, command name, and command parameters. Each segment must conform to the format stated in the specification. When reading a command name, the parser reads until either a left parenthesis or end-of-command character is recognized. Blank spaces in between the characters are allowed but ignored. The command parameter parser is a generic one that handles all the commands, hence, it accommodates variable length commands. For example, a "home" command contains no parameter and a "move axis" com-

mand can contain one, two, or three axis parametric values.

The testing utilities also contain a set of common command classes for each of the commands (the common command and response classes are described in Section X-B). Once the parser verifies the command name, the corresponding command class will be used to store the associated parameters. Methods relating to each command are predefined in each command class. A variable-length linked list stores the valid input command class instances in the order that they are received. Currently, if the CMM utility receives an "abort" command, it will clear the queue (as required in [3]).

At the end of the parsing, the CMM utility sends a confirmation signal back to the application. The confirmation conveys a message that the input is either a valid or erroneous command. The parser contains a function to determine the severity of the errors.

We partition the CMM utility software such that users can either take the parser and integrate only that into their own machine controllers or also use the execution and simulation modules that are a part of the CMM utility software.

The execution of each of the commands has been implemented using a state machine model. At the beginning of the state machine, a new-command flag is verified and, if true, a new instance of the command class will be generated and populated with the information that is specific to the command that is just retrieved from the command queue. A new instance of the response class will also be generated and the attributes updated throughout the execution. The state machine model has additional common features for all the commands. It checks for certain common errors, such as "CMM not initialized" and "invalid command parameters." After the errors are checked out, the state machine calls the simulator for the command.

B. Command and Response Classes

An important part of the testing utilities is a common set of command and response classes. Both application and CMM utilities use this common set of definitions for command and response. Using these (or similar) command and status classes in implementations of either specification [3][4] will reduce development and debug time and will streamline the testing and analysis process. NIST has developed a set of CMM-driver specification compliant command and response classes which are currently being used within the CMM utility software and a CMM-driver specification compliant version of the NIST DMIS interpreter. The command and response classes are written in C++. Accompanying these classes are C++ files defining various methods for each of these classes. In order to simplify the class structure, the actual command and response classes are derived classes from command and response base classes, respectively. The primary function of these classes is to provide a common set of data structures for passing data and generating command and response strings. The role of the command and response classes is illustrated in Figure 4. Also defined are classes for handling data types and errors defined with in the specification. The data classes contain the necessary logic for formatting the data per the specification when a command or response string is being generated. Figure 10 gives an example of command and response (status) classes supporting the CMM-driver specification (this specification does not specify the structure of the classes).

C. Analysis Tools and Metrics

In order to better utilize conformance testing, quantitative metrics are essential. This will give all implementors measurable incentive to perservere until their implementation receives high marks in all tests. It will also help as a presentation tool to management in all the organizations involved in testing, to quantify progress with simple graphs. Several spreadsheets have been developed. An example spreadsheet, shown in Figure 12, gives some preliminary metrics for measuring CMM implementations tests, such tests as are illustrated in Figures 5 and 6. These spreadsheets need to be completely filled in for all test cases.

The data analysis tool currently extracts communication performance information from the data log files that both the application and CMM utilities generate. The analysis tool generates a performance report that consists of two parts, for each individual command and for the command file. For each individual command, the analysis tool indicates whether the command has been handled at every stage and within user specified timing ranges, and whether an error condition has occurred. The command performance for the following critical stages were listed:

1. command sending, whether and when (referring to the execution starting time)
2. command receiving
3. command acknowledgement signal sending
4. command acknowledgement signal receiving
5. command completion signal sending
6. command completion signal receiving.

The summary part of the performance report totals the numbers of commands that are properly and not handled properly, including not received and taking too long to respond. The performance report is illustrated in Figure 11.

XI. Testing Procedures

Validation (functional) testing and conformance testing of any specification are both essential. Neither formal validation testing nor formal conformance testing has yet been done on either specification. The implementation and testing that has been done on the CMM-driver specification, however, has provided informal validation and conformance testing. Informal testing should continue and formal testing should begin. It will

```
Command
Description:  Probe to absolute position 10,20,30 on the XY plane.
Implementation Code:
_probeToCommand = new probeToCommand(ABSOLUTE);
_probeToCommand->setAxis(X,10.000);
_probeToCommand->setAxis(Y,20.000);
_probeToCommand->setAxis(Z,30.000);
_probeToCommand->setUnitSurfaceNormal(0,0,1);
cout << _probeToCommand->getCommandString() << endl;
delete _probeToCommand;
Output:  PROBE_TO(X =10.000, Y= 20.000, Z = 30.000, 0.000, 0.000, 1.000)

Response
Description:  Actual position probed 10.002, 19.999, 30.001 on the XY plane.
Implementation Code:
_probeToStatus = new probeToStatus(++id);
_probeToStatus->setAxis(X,10.002);
_probeToStatus->setAxis(Y,19.999);
_probeToStatus->setAxis(Z,30.001);
_probeToStatus->setUnitSurfaceNormal(0,0,1);
cout << _probeToStatus->getStatusString() << endl;
delete _probeToStatus;
Output:  TOUCH(X =10.002, Y = 19.999, Z = 30.001, 0.000, 0.000, 1.000)
```

Fig. 10. Example command and response (status) class usage

```
Starting time: 01:07:11_15:44:49:60

CMD NAME                          CMDSENT      CMDRECEIVED ACKSENT  ACKRECEIVED COMPLETIONSENT   COMPLETIONRECEIVED
1 INIT_DRIVER()                   15:44:49:603  0.301 sec    0.0 sec   0.0 sec     0.0 sec          0.120 sec
2 SET_AUTO_MODE()                 15:44:50:354  0.50 sec     0.0 sec   0.0 sec     0.0 sec          0.120 sec
3 GOTO(142.756,127.671,-73.809) ) 15:44:51:105  0.301 sec    0.0 sec   0.10 sec    1.492 sec        0.120 sec
...
47 MOVE_HEAD(2,90,90)             15:45:24:153  0.300 sec    0.0 sec   0.0 sec     34.550 sec ^^long ^^ 0.0 sec
...

SUMMARY:

CMD file name: lego1.prg
TOTAL NUM OF CMDS IN FILE: 60
TOTAL EXECUTION TIME: 0 hour(s) 1 minute(s) 27.578 second(s)

CMDS PROPERLY RECEIVED AND EXECUTED: 60
CMDS NOT RECEIVED BY RECEIVER: 0
CMDS TOOK TOO LONG TO BE RECEIVED BY RECEIVER: 0

ACK NOT RECEIVED BY SENDER: 0
ACK TOOK TOO LONG TO BE RECEIVED: 0

COMPLETION NOT RECEIVED BY THE SENDER: 0
CMDS TOOK TOO LONG TO ACCOMPLISH: 2
```

Fig. 11. Text output log of the analysis tool

be useful if an organization consisting of NIST and vendors from both sides of the interface is formed to collaborate in testing. A procedure or process is needed to guide the conformance testing to achieve testing efficiency and high quality implementations.

A. Validation Testing Procedures

Informal validation occurs naturally as part of implementation. For example, when an application vendor writes the software that generates commands provided in the specification, if the vendor discovers that there is

Common application test software with CMM					
CMM type:					
Test case execution in simulation	Did the plan execute to completion? (1 for yes, 0 for no)	If the plan executed to completion, how many lines did not match the benchmark output file?	If the plan did not execute to completion, how many lines did your system execute before failure?	If the plan did not execute to completion, how many hours of code rewriting were required to fix the problem?	If the plan executed to completion, but one or more lines of output do not match the benchmark output file, how many hours of code rewriting were required to fix the problem?
#1					
#2					
#3					
#4					
#5					
#6					
…					
Test case execution on CMM	Did the plan execute to completion? (1 for yes, 0 for no)	If the plan executed to completion, how many lines did not match the benchmark output file?	If the plan did not execute to completion, how many lines did your system execute before failure?	If the plan did not execute to completion, how many hours of code rewriting were required to fix the problem?	If the plan executed to completion, but one or more lines of output do not match the benchmark output file, how many hours of code rewriting were required to fix the problem?
#1					
#2					
#3					
#4					
#5					
#6					
…					

Fig. 12. Example metrics spreadsheet for testing CMM implementations of the specification

no command to perform a required action, the vendor raises an issue against the specification.

Validation testing is also a side-effect of conformance testing. If a vendor attempting to build software that conforms to the specification finds that it is unreasonable or impossible to make the software conform, the vendor raises an issue against the specification.

It will be useful to devise a procedure for collecting issues discovered during implementation and conformance testing and considering whether and how the specification might be changed. The Standard Improvement Request process used by the DMIS National Committee provides a model of how this might be done.

Formal validation testing is conducting by identifying functional scenarios that should be supported by a specification and trying to carry out each scenario using the specification. The total set of functional scenarios should cover the entire range of functionality expected to be supported by the specification. This set should be agreed on by the community interested in the specifica-

tion.

Scenarios might be given in natural language, such as "inspect hole A and plane B on the part whose design is in CAD file C," or they might be given in inspection programs written in DMIS or some other high-level language. Manual use scenarios should also be included, since the specification needs to support manual use.

The role of NIST in validation testing has not yet been determined, but it is likely to include helping devise validation tests and conducting validation testing.

As part of validation testing, it is essential to conduct interoperability testing like that shown for conformance testing in Figure 7. In interoperability testing, each application tries working with several CMM vendors and each CMM vendor tries working with several applications. At the stage where software on side A of the interface has been made to work with only one vendor's software on side B of the interface, it is nearly certain that the side A software will not work with a second vendor's side B software. The failures are most likely to reveal that one side or both are not actually conforming. The failures, however, are also likely to reveal ambiguities or other inadequacies in the specification and inadequacies of conformance tests. These should be reported when discovered.

B. Conformance Testing Procedure

A conformance testing procedure is meant to test the compliance of an implementation to the specification. Metrics and analysis are needed to determine the degree of compliance. Conformance tests may be conducted privately by a vendor or they may be conducted by a testing service set up to do conformance testing and issue certificates (or other assurances) of conformance. At one extreme, a conformance testing service might make detailed results of all conformance tests publicly available. At the other extreme, a conformance testing service might make nothing public except certificates of conformance. Test results and trends may be performed privately or at some independent site. The degree of openness is determined by what users and vendors prefer. A side effect of the conformance testing process is to effect an improved interface specification, which can ultimately lead to quantifiable CMM system interoperability results.

The test files and procedures used by a conformance testing service should be publicly available. Ideally, a vendor will have run all the tests and adjusted the software so that it passes all the tests before submitting it to a conformance testing service. Large users might conduct conformance tests themselves. It is expected that particularly interoperability testing will be performed in a distributed manner. Potential commercial implementations of the specification need to be involved in the testing. Care must be taken to insure that testing results are not made available to unwanted parties.

The role of NIST in conformance testing is to devise testing files and procedures that may be used by testing services, vendors and users.

C. Collaborative Testing

It will be useful to form an organization of representatives from NIST and metrology systems vendors (on both sides of the interface) to do informal validation and conformance testing. We assume here that this organization also is responsible for development and modification of the specification; if that is not the case, procedures for changing the specification will be less direct than described below.

We envision two or three phases for this procedure. Phases one and two are differentiated by an expanding scope of commands. Phase three (not yet fully defined) expands to more devices accompanied by a formalization of the specification into a standard. Metrics that will be the measure of testing success are discussed in X-C.

C.1 Phase One Testing

Phase one scope would include some of the following commands:

- two degree of freedom (DOF) probe head and three DOF move of CMM arm (five DOF total)
- measuring in same five DOF
- abort
- get and set parameters

Phase one will exclude error correction and recovery, scanning, rotary motion, and probe calibration. Among other benefits, this will help to debug the conformance testing procedure itself. Phase one testing will consist of tests on the various CMMs and/or their simulators as well as tests on third party inspection program execution software, independently and together. Various high level languages, *e.g.*, DMIS, inspection plans will be required, files of command strings in the format of the emerging specification, and response rules files. Software tools agreed to for use by all participants must be employed in testing to minimize variability.

C.2 Phase Two Testing

The scope of phase two will include error correction and recovery, scanning, rotary motion, and probe calibration as well as the entire set of commands in the specification.

C.3 Conformance Testing Procedure for Both Phases

1. Design (or select or modify) and agree upon an appropriate artifact or artifacts.
2. Develop and achieve consensus on appropriate high level inspection programs and specification compliant command files for each chosen artifact, and response rules for the CMM utility software.
3. Develop, collect, and publish on the web any necessary files (*i.e.*, DMIS files, specification command files,and response rules) along with file format

specification (allowing automated output file validation) that will be the standard baseline for testing. Specification-compliant command files will be used in testing because we will not be able to encode in certain high level languages certain behaviors we can perform with CMM driver commands. Furthermore, high level to low level language interpreters will not, in general, be able to produce certain important types of errors.

4. Develop and publish on the web performance analysis tools consisting of various metrics of performance and reporting facilities. These tools will be used to quantify the success of the test results. All participants will agree to these metrics before testing begins.
5. Using the specification compliant inspection programs and application utility software, vendors and users will implement the specification, conduct tests, and do performance analysis.
6. CMM utility with response rules: Using the NIST test suite, third party inspection program software vendors and third party software users will conduct tests and do performance analysis.
7. Integrated system testing: Using high level inspection plans and test artifacts, application software vendors, application software users, CMM vendors, and CMM users will conduct tests on an integrated system and collect data. These tests may be done remotely in a distributed manner, assuming that high level programs require no real-time interaction with the CMM from command to command.
8. Performance results will be collected, stored, and analyzed using agreed performance metrics. Results of this analysis will be made known only to participants, their managers, and any oversight committee.
9. An issues log will be maintained and all participants will log issues resulting from the testing. Regular conference calls following testing sessions will be centered on the resolution of these issues.
10. Modify the specification as needed.
11. Modify the common utility testing software as needed.
12. If the specification is stable or a previously determined time period has expired, conclude and move to next phase. If the specification is not stable, go back to the first step in this procedure.

XII. Real-Time and Object-Oriented Issues

There are at least two levels of real-time in this testing procedure. In the narrow sense, we are testing a real-time system remotely over a non-real-time link. In the broader sense, the specification must be tested in "real-time," though now the time scale is much longer.

In the narrow sense of real-time, we cannot do remote testing over a non-real-time link unless commands in the specification are completely independent in time, so that the success of a command, or series of commands, is not at all dependent on the relative time any previous or subsequent command is executed or completed. For example, the definition of the "move" command will determine the timing requirements for sending commands. If "move" specifies a non-zero final velocity of the probe head, the system is now dependent on the timing of the subsequent command(s). However, this is not a problem if commands are allowed to be queued in the CMM implementation. The CMM-driver specification allows such queueing, and it implicitly requires it of the CMM implementation. Therefore, assuming that all application implementations expect command queueing in the CMM implementation, the application will send subsequent commands upon receipt of an "OK" response only *versus* "OK" and "done" or "error." The CMM implementation can then look ahead and wait to execute all adjacent "move" commands together which will allow non-zero final velocities for some of the "move" commands. The CMM-driver specification allows "fly mode," *i.e.*, non-zero final velocities, only for adjacent "move" commands.

In the broader sense of real-time, the specification must be developed, implemented, and tested successfully by a broad majority of vendors and users within a certain time frame that might be defined as the technology change window, *i.e.*, the specification cannot be obsolete by the time it appears in new products.

The issue of objects has been in the forefront of development in both specifications. The I++ specification has chosen to make objects (in the form of Unified Modelling Language (UML) models) as part of the specification, whereas the CMM-driver specification makes no such requirement. However, we argue for a middle ground where the object models are part of the testing and implementation tool set that allow implementors to facilitate implementation, testing, and integration. This will both facilitate implementation development (and subsequently interoperability) and will allow implementors to choose non-object oriented implementations while still complying with the specification.

XIII. Distributed Real-Time Tests

A real-time, distributed, and cross-continental demonstration was conducted on July 13, 2001 in which a CMM-driver specification compliant file was used as input for sending commands to LK Metrology in Darby, United Kingdom (UK) from the National Institute of Standards and Technologies (NIST) in Gaithersburg, Maryland, USA, where the commands were used for inspecting the test artifact of Figure 8. A user-controlled pan/tilt/zoom camera was also integrated into the environment with a web-based video server to allow the inspection program to be viewed at either location. The demonstration used only one inspection plan test file.

This demonstration helped us improve NIST testing utilities and showed that distributed testing is not only possible but efficient and beneficial to any future testing events.

XIV. Future Work

Depending on the needs of the CMM industry, we may create and perform explicit validation tests. Additionally, more precise metrics need to be defined in concert with the CMM community of users and vendors. Our current set of metrics (Section X-C) is preliminary only.

The current set of testing utilities is focused on the CMM-driver specification detailed in [3]. Based on a recent study and comparison of the two specifications [5], it is clear that the two specifications have more similarities than differences. It currently appears that the community of CMM system users and vendors is moving to integrate the two specifications into one while using the I++ specification [4] as the baseline. It seems clear that the next step is to perform any modifications necessary to make the current testing utilities compatible with the I++ specification.

Perhaps the most important single item lacking in our test suite is to develop a comprehensive response simulator for use in application implementation testing. This may also be the most challenging part of test suite development. We also need to discern whether to couple our response simulator (which includes the response rules) with high-level test inspection plans, *e.g.* in DMIS, for use with the CMM utility test software. The questions and problems associated with this decision are discussed in Section X-A.4.

References

[1] ISO 14750:1999, *Interface Definition Language*, ISO, Geneva, Switzerland, 1999.

[2] ISO 10303-11:1994, *Industrial automation systems and integration - Product data representation and exchange - Part 11: The EXPRESS Language Reference Manual*, ISO, Geneva, Switzerland, 1994.

[3] David Smith [LK Metrology], Lutz Karras [Zeiss IMT], Michel Penlae [Xygent], and William Wilcox [Wilcox Associates], "CMM-driver Specification," Release 1.9, 2001.

[4] Hans-Martin Biedenbach [Audi], Josef Brunner [BMW], Kai Gläsner [DaimlerChrysler], Günter Moritz [Messtechnik Wetzlar], Jörg Pfeifle [DaimlerChrysler], and Josef Resch [Zeiss IMT], "I++ DME-Interface," Release 0.9, 2001.

[5] Thomas Kramer and John Horst, "A comparison of the CMM-driver Specification Release #1.9 with the I++DME-Interface Release 0.9," 2001.

[6] "Revised report on the algorithmic language algol 60," *Communications of the ACM*, vol. 3, no. 5, pp. 299–314, May 1960.

[7] Consortium for Advanced Manufacturing - International, *Dimensional Measuring Interface Standard*, Revision 3.0, ANSI/CAM-I 101-1995, CAM-I, Arlington, Texas, 1995.

[8] ISO 10303 Part 1:1994, *Industrial automation systems and integration - Product data representation and exchange - Part 1: Overview and fundamental principles*, ISO TC184/SC4, ISO, Geneva, Switzerland, 1994.

[9] Martha Gray, Alan Goldfine, Lynne Rosenthal, and Lisa Carnahan, "Conformance Testing," *http://www.itl.nist.gov/div897/ctg/conformProject.shtml*, 2000.

[10] Thomas Kramer, "Feature-Based Inspection Control System (FBICS)," 2002.

[11] ISO 10303-224:1998, *Industrial automation systems and integration - Product data representation and exchange - Part 224: Application Protocol: Mechanical Product Definition for Process Planning Using Machining Features*, ISO, Geneva, Switzerland, 1998.